AMENDMENT OF TITLE

EVENT-DRIVEN QUEUING SYSTEM AND METHOD FOR ALLOCATING COMPUTER RESOURCES

AMENDMENT OF DESCRIPTION

[3.]   A "queuing system" is software **running on computer resources** that allocates the processing of jobs among networked computers, issues commands to such networked computers, and tracks the jobs through completion; many queuing systems generate status and/or error reports as part of job tracking. The detailed nature of "processing a job" varies by industry, but is generally a computational task that often consumes computing resources for an extended period (minutes to hours, or even days). Queuing systems exchange messages with users and with processes running on resources, such as computers, storage devices, networks, network devices, and associated software (collectively, "resources"). A "host" is a computer connected to a network and capable of running application software. The population of hosts connected to a common network and available to perform processing is called a "farm". A "platform" is a computer with a given combination of operating system, central processing unit ("CPU"), and related hardware specific to such CPU.

[11.]   To understand the improvements in queuing system performance provided by the EDQS invention, an overview of how queuing system performance is analyzed is helpful. Queuing system performance ("T(n)") is approximated by the "sort and dispatch" equation, $T(n,m) = (n * m) + (n \log n) + (m \log m)$, where "n" is the number of jobs, "m" is the number of processing hosts, "(n log n)" is the time required for a job sort routine, "m log m" is the time required for a host sort routine, and "n * m" is the time required for match jobs and hosts. Existing art queuing systems build a master job queue, a master host queue, and then a master job **dispatch** queue by matching job and host. The cumulative time to sort and dispatch is called the job dispatch time. Most queuing systems run the sort and dispatch routine periodically, often called a "time out", "collect and analyze", or "fixed period" approach. Using a single processing thread in the management daemon, job submissions are typically "insert sorted" (the "n log n" component) upon receipt to produce an updated job queue. The hosts are then sorted and matched with jobs (the"(n log n)" and "(n*m)"components, respectively) to build a master job **dispatch** queue. The host sort and match computation uses a separate processing thread that runs periodically. Queuing performance can be improved by "filtering" to delete jobs already being processed and

to delete unavailable hosts before running the sort and dispatch routine, which produces the revised equation: $T(n,m) = [(n - i) * (m - j)] + [(n - i) \log (n - i)] + [(m - j) \log (m - j)]$, where "i" is number of jobs removed from list, and "j" is the number of hosts removed from list. There are typically more jobs than hosts, so the job sort routine typically takes more time than the host sort routine.

[12.]   Even optimized by filtering, the existing art of sort and dispatch has at ~~last~~ __least__ seven major problems: (1) the job sort routine could take longer than the interval between periodic host sort routines, called a job sort overrun condition, which is normally fatal to dispatch; (2) the sort and dispatch routine is run periodically, even if unnecessary, which can result in delays or errors in completing other supervisory tasks, e.g., missed messages, failures to reply, and hung threads; (3) an available host may experience excessive delay before receiving a new job because of the fixed interval on which the sort and dispatch routine runs, called a "host starvation" condition; (4) the sort and dispatch routine is asymmetric and must be executed on a single processing thread; (5) the number of jobs that a queuing system may reasonably handle is limited strictly by the amount of time it takes to execute the sort and dispatch routine; (6) the existing art produces uneven service, particularly erratic response times to job status queries by end-users and erratic runtimes; and (7) uneven service offends and provokes end-users.

[13.]   The speed at which a management daemon in a queuing system is able to match a job with a host and to dispatch the job for processing is the key to the performance of a queuing system. It is also very desirable for a queuing system to gracefully handle, without system administrator involvement, the insertion of "deadline jobs" into the system. Because most queuing systems use a fixed period scheduling algorithm (aka "fixed period queuing system"), it is impossible for them to easily accommodate deadline jobs or "sub-period jobs". A "sub-period job" is one that takes less processing time than the time it takes to schedule the job for processing. A fixed-period queuing system may take note of a job submission data upon receipt, but typically builds a master job __dispatch__ queue periodically, as explained above; only after a master job __dispatch__ queue is built can the management daemon dispatch a job to a host for processing. Another drawback of fixed-period queuing systems is that a job sort and a host sort require a significant amount of processing power; and dispatch times increases exponentially for an arithmetic increase in jobs or hosts. ~~This~~ __These__ are very serious problems for large farms.

[19.]   The Event-Driven Queuing System and Method ("EDQS") invention uses a continually updated, event-driven, supervisory daemon together with one or more callback, event, process groups, job typing, message architecture, clients, workers, executors, and related functions, as described below, to solve the above-described problems in the existing art. The invention embodies innovative steps in (1) how messages are constructed to enable execution on heterogeneous platforms, (2) what messages need to be passed, between what processes or users, and at which points in a job's processing, (3) the level of granularity needed for a particular decision and for messages related to that decision, (4) how to dispatch jobs without building a master job **dispatch** queue, and (5) how to build and continually update a supervisory daemon responsive to events and triggers characteristic of farms. In its preferred embodiment, the EDQS is built with object-oriented programming ("OOP") code. OOP terms well known in the art, such as object and container **(i.e., a class of objects that contain other objects)**, are used herein. The EDQS invention can be built and implemented using other programming language models that provide functionality equivalent to that described herein.

[020.] The job allocation and management component of the supervisory daemon of the EDQS invention maintains an inventory of all resources available for assignment for processing jobs. A "job" in the **preferred, OOP embodiment of the** EDQS invention is a container that contains the parameters required to execute a task on a host. The term "job" is often used herein as short-hand to mean the task to be executed. Where appropriate, the phrase "job container" is used to distinguish the OOP container from "job" in the sense of "task to be executed". **In non-OOP embodiments of the EDQS invention, a file directory or folder replaces an OOP container.** The term "job package" means the "package information", i.e., the information required to execute the task; job package has a defined data structure. The term "supervisor" means a host that runs, among other software, a "supervisory daemon," as explained below. The term "worker" means a host on which a job is processed. A worker in the EDQS invention runs, among other software, a "worker daemon" and one or more "executors," as explained below.   "User" means either a human end-user ("end-user") or a process. The EDQS supervisory daemon ("supervisory daemon") may be hosted on its own server or another server in the farm network.

[28.]   The EDQS invention allows a system administrator to define the behavior of a queuing system in three different ways: (1) interactively, by manually manipulating the priority of the jobs, and variables in the algorithms applied to match jobs to workers; (2) dynamically, by having the system itself ~~to~~ autonomously respond to events triggered by the processing of jobs, and (3) adaptively, by being able to change the behavior of the system based on evolving conditions without user intervention.

[62.]   The following are some of the defined terms used to describe the EDQS invention.

A "client" is the combination of a user and a user interface.

A "daemon" is a program that runs continuously and exists for the purpose of handling periodic service requests that a computer system expects to receive. A daemon forwards the requests to other programs (or processes) as appropriate.

An "executor" is a procthread launched by a worker as a result of running a job type executable.

A "job", **in OOP embodiments of the EDQS invention,** is a container that contains the parameters required to enable a task to be executed on a worker and the course of events related to the submission and processing to be tracked **(In non-OOP embodiments of the EDQS invention, a file directory or folder is used instead of a job container.).**   In a preferred embodiment, a job contains the job name, a job serial number ("job ID"), job requirements, job restrictions, job reservations, job package, job agenda, and the job typename. The "package" is a pointer to an arbitrary data structure that can be used to pass job information from the submission side to the execution side. The "typename" is simply a tag that describes what kind of job type to execute. The executor looks at the job type stored configuration file for the specified typename to retrieve the available options in launching the job. The data in the job type configuration specifies to the executor what executable to run, and what language binding to use in order to run the executable. A job always contains at least one sub-job. Each job container also contains a callback table and a callback helper table, and may contain an agenda table, ~~and~~a subjob table**, and other data, as described in the "EDQS event/callback architecture" section below.** The subjob table is used to

manage the individual instances of a job in each individual worker to which a part of the job is assigned, for instance, when a job to render a series of animation frames is processed by more than one worker. Typically, all application software used to process a source file is entirely resident on a worker, and the supervisory daemon does a preliminary match of job and worker in a dispatch operation. In the preferred embodiment, the ultimate decision on suitability of ~~an~~ a worker rests with the process spawned by the "check environment" executable specified in the job. Because the worker doesn't completely describe the execution environment of a worker, the check environment process spawned by the job determines if the execution environment of a given worker is completely adequate. Most of the time, a proper match between job and worker is made by the supervisory daemon, especially in embodiments in which a user can restrict during job submission the eligible workers by use of the restrictions attribute in the job container. The term "job" is used herein as short-hand to mean the course of events related to performing a task on a worker. When necessary to distinguish the job container from components related to processing, the specific terms (e.g., job container, etc.) are used. In alternative embodiments, a job container also contains OOP method executables related to routing and/or processing a task. When "job" is used herein in connection with existing art queuing systems, job has an existing art meaning of a task to be performed on a candidate. To eliminate the need to set up special cases when there are no subjobs, job containers with just one job typically denote the job as a subjob.

[63.]   A "management daemon" is daemon responsible for scheduling and keeping track of all jobs in existing art queuing systems.

An "optimal match" is a system-defined variable used in comparing jobs and workers; the comparison variable typically used ~~in~~ is worker's CPU speed, with faster being better. Other worker capabilities variables can be used as ~~the~~ a comparison variable, e.g., file system, RAM, job processing application.

[66.]   Next, the supervisory daemon requests that the database manager for the worker list database approve the request. **In the time required to select the preferred worker, the idle worker list may have changed.** The database manager **therefore** locks the worker list database and searches for **the** preferred worker. If **the** preferred worker is on **the** list

and is available, the database manager grants the request, marks the preferred worker as
unavailable on the list, and unlocks ~~it~~ **the list**; if the preferred worker is not on **the** worker list
when the request is received, the database manager denies the request, and unlocks list. If the
database manager approves the request, the supervisory daemon exchanges messages with the
preferred worker to confirm that adequate capabilities are currently available to process the job,
and if the availability of worker capabilities is confirmed, the supervisory daemon informs the
check environment process of approval and dispatches the job to the preferred worker; if the
worker capabilities are not confirmed, the supervisory daemon denies the request from the check
environment process and instructs the process to continue comparing entries in the eligible
worker list (unless the system-defined limit for increment comparisons has been reached, as
described below). If the database manager denies the request for the preferred worker, the
supervisory daemon denies the request from the check environment process and instructs the
process to continue comparing entries in the eligible worker list (unless the system-defined limit
for increment comparisons has been reached, as described below).

[67.]   The eligible worker list has not been sorted at this point, which saves the time and
processor resources required to sort workers. If the embodiment does not use the check
environment process, the worker commences processing the job, and if the processing fails, such
failure constitutes the worker rejecting the job; (6) if the check environment process returns a
Boolean false, the first entry on the eligible worker list is rejected (the "worker rejects the job",
since the check environment process runs on the worker), the supervisor receives messages from
the worker that the worker has rejected the job, the supervisor dispatches the job to the next most
optimal worker based on comparison, and the check environment process analyzes that
~~workers~~ **worker's** capabilities and properties, and so on until a match of job and worker is
made, or a system-defined limit is reached, typically n/2 comparisons of job and worker, where
"n" is the number of jobs in the eligible worker list. An entry on the eligible worker list might be
rejected if the job actually needs a higher value of an attribute than is specified in a given
worker's attributes, e.g., a job might need the latest version of a given application program, and
only the check environment process can confirm that the worker has the correct version; (7) if a
match of job and worker has not been made after n/2 comparisons of job and worker, the
supervisory daemon sorts the remaining workers (i.e., the workers for which a job to worker
comparison has not been attempted) to produce a sorted list (i.e., a queue) of workers in order of

one or more attribute fields, e.g., CPU speed, RAM, software version. There may be multiple workers aliased to a given worker, but the processor speeds, RAM, and rendering program versions may vary above the minimum values that define the suitable worker for a given node; and (8) after a sort of the remaining workers in the eligible worker list, the supervisor dispatches the job to the top-ranked worker and the steps from 5 through 8 repeat until a match of job and worker is made or the list is exhausted. If the list is exhausted, the job waits for a worker-driven dispatch. Importantly, the restrictions contained in a job type can be applied in step no. 2 to produce an eligible worker list that conforms to such restrictions. The job's reservation attribute is used by the supervisory daemon to decrement the amount of available capabilities on a worker when processing of the job starts on the worker.

[68.]   To summarize CSPS job-driven dispatch steps:

Step 1. The worker list is filtered for availability to produce the idle worker list.

Step 2. The idle worker list is filtered using capabilities and requirements, restrictions, and reservations to produce the eligible worker list.

Step 3. The eligible worker list is searched, without sorting, for the ~~closest~~ **first** match based on capabilities vs. requirements.

Step 4. The worker is then reserved in the worker list database. (Typically, this uses the record lock/unlock feature of a database management system.)

Step 5. The supervisor dispatches the job to the worker.

Step 6. In the preferred embodiment, the worker checks the worker environment using the check environment process.

Step 7. The worker accepts or rejects the job. If the worker accepts, the worker processes the job. If the check environment process is not used, the worker commences processing the job, and if the processing fails, such failure constitutes the worker rejecting the job.

Step 8. If the worker rejects the job, messages to that effect are exchanged between the worker and the supervisor, the supervisor removes the worker from the eligible ~~work~~ **worker** list, the supervisor goes to step 3, and will loop back from step 3 to step 8 up to **the system-defined limit, e.g.,** n/2 times, if the job rejects the worker in step 7 in a given iteration.

Step 9. If, after **the system-defined limit (e.g.,** n/2**) of** tries to match a job with a worker there is still no match, the supervisor sorts the remaining workers and systematically

attempts to dispatch the job to the first ranked worker, and if rejected, to the second ranked worker, as so on until there are no workers remaining on the list.

Step 10. If the eligible worker list is exhausted, the job waits for a worker-driven dispatch.

[070.] Next, the supervisory daemon requests that the database manager for the worker list database approve the request. The database manager locks the worker list database and searches for **the** worker and grants the request, marks the worker as unavailable on the list, and unlocks the worker list. If the database manager approves the request, the supervisory daemon exchanges messages with the preferred job to confirm that adequate capabilities are currently available to process the job, and if the availability of job capabilities is confirmed, the supervisory daemon informs the check environment process of approval and allows processing to begin on the worker; if the job capabilities are not confirmed, the supervisory daemon denies the request from the check environment process, instructs the job to wait, and goes to step 3 above to try to find another job for the worker (unless the system-defined limit for incremental comparisons has been reached, as described below).

[071.] The eligible job list has not been sorted at this point, which saves the time and processor resources required to sort jobs. If the embodiment does not use the check environment process, the worker commences processing the job immediately after confirmation of worker capabilities, and if the processing fails, such failure constitutes the worker rejecting the job; (6) if the check environment process returns a Boolean false, the first entry on the eligible job list is rejected (the "worker rejects the job"), and the check environment process analyzes the second entry on the eligible job list, and so on until a match of worker and job is made, or a system-defined limit is reached, typically m/2 comparisons of worker and job, where "m" is the number of workers in the eligible job list. An entry on the eligible job list might be rejected if the worker actually needs a different value of an attribute than is specified in jobs on the eligible job list at the time (only the check environment process, or when the check environment process is not used, attempting to process the job, can confirm that the job has the correct version); (7) if a match of worker and job has not been made after **the system-defined limit (e.g.,** m/2**) of** comparisons of worker and job, the supervisory daemon sorts the remaining jobs (i.e., the jobs for which a worker to job comparison has not been attempted) to produce a sorted list (i.e., a queue) of jobs in order of one or more attribute fields, e.g., software version; and (8) after a sort

of the remaining jobs in the eligible job list, the supervisor dispatches the first ranked job to the worker, and steps 5 through 8 are repeated until a match of worker and job is made or the list is exhausted. If the list is exhausted, the worker waits for a job-driven dispatch. The ~~worker's~~ **job's** reservation attribute is used by the supervisory daemon to decrement the amount of available capabilities on a ~~job~~ **worker** when processing of the ~~worker~~ **job** starts on the ~~job~~ **worker**.

[74.]   In a job dispatch, the message exchange between the worker daemon and the executor begins with the receipt of a dispatched job ("Start Job" message) from the supervisory daemon. Upon receipt of the Start Job message, the worker is given option to accept or reject the order. The conditions in which this occurs are based upon whether the worker can satisfy the requirements of the job and comply with the job's reservations attributes. In the preferred embodiment, the check environment process executes on the worker and determines whether the job should work properly. If the check environment process approves the worker, an instance of the executor is launched with the job's ID and subjob ID as parameters. The worker daemon reverts **to** listening for network messages and the executor processes the job, as follows. The executor decodes the job ID and subjob ID, then uses this information to send a network query to the worker that hosts the executor (the "launching worker"). In this query, the executor requests, and the worker obtains and relays **to the executor**, additional data about the job corresponding to the job ID and subjob ID. The additional data includes the job type of the job and specifies which interpreter the executor should launch. Based on the additional data, the executor selects the proper bootstrapping method that will present the job in a format the interpreter can interpret. For example, the bootstrapping method generates a Perl script that contains all the retrieved job data. The Perl interpreter is then launched and initialized with script generated by the bootstrapping method. The interpreter executes the script, which launches the job under the control of the executor. The executor then contacts the worker daemon and reports that the job, identified by job ID and subjob ID, is to be marked as "running". In the event a problem is detected, e.g., job process crashing, it is the executor's responsibility to detect and report to the worker daemon a "failure." The executor typically sends heartbeat packets to the worker to signal that the executor and its job have not crashed. In the event the

executor crashes, its heartbeat packets stop, and the worker daemon assumes the job has failed, marks the job as failed, and sends a message to the supervisor to that effect. Each worker sends heartbeat packets to the supervisor to signal that the worker has not crashed. The worker periodically sends job status messages to the supervisor so that the supervisor can track workers and jobs.

[75.]   The CSPS method normally matches a job and a worker on the first try. In the worst case search scenario, the supervisory daemon searches through the entire list of available workers n times. To balance the risk that a linear search will not find a suitable idle worker or job, as the case may be, versus the cost of the search, the linear search is **typically** only repeated up to n/2 **or m/2** times. If the job is not dispatched after n/2 searches **in a job-driven search**, **or a worker does not receive a job after m/2 searches in a worker-driven search,** the supervisory daemon prepares and executes a sort. So, at best, the performance after filtering and using a linear search of suitable workers is O(n), and in a worst case, performance is O(n.sup.2).

[76.]   If the system-defined limit for number of comparisons during a linear search is reached (typically n/2 or m/2 comparisons), the CSPS method sorts the jobs and workers as follows. (If the system-defined limit for number of comparisons is set to "n", the sort is not performed.) To prepare for a sort, the ~~CSS~~ **CSPS** method first compares a given job, worker by worker, (or vice versa in a worker-driven sort) to produce weights for each combination of job and worker. In the preferred embodiment, the CSPS sort is run only as part of a job-driven or worker-driven dispatch, and only then after a linear search has reached a system-defined number of comparisons. Depending on policy, typically set by the system administrator, all workers or a subset of all workers, and all jobs or a subset of all jobs, can be included in a CSPS sort. Jobs and workers are typically filtered based on availability, job attributes, and worker attributes, to produce an eligible worker list in a job-driven sort, or an eligible job list in a worker-driven sort. After a sort, the resulting queue is used as the basis for dispatch, as described above. In other words, if n/2 workers were sorted in a job-driven dispatch, the comparison of job with worker resumes at the top ranked worker in the queue; if ~~n/2~~ **m/2** jobs were sorted in a worker-driven dispatch, the comparison of job with worker resumes at the top ranked job in the queue.

[80.]   The CSPS comparison method quickly determines how closely Job A, submitted using Job A's home node, matches a given node, versus how well Job B, submitted using Job B's home node, matches the same node. The normal increment of weight, when added, is 1. A comparison of the weights accorded each job allows the final result of equality or inequality to be determined numerically, within two machine instructions in most cases. After each job has been assigned its weight on each node, any sorting algorithm known in the art, such as bubble sort, insertion sort, or QuickSort, may be applied to sort jobs with reference to a given node by comparing job weights. The preferred embodiment uses QuickSort, (available, e.g., at ~~www.nist.gov/dads/HTML/quicksort.html~~ **the National Institute of Science and Tecnology website, search on "quicksort"**). The results of the illustration would be that Jobs A and B are equally weighted on all nodes other than their home nodes; on /project1, Job A weight is greater than Job B; and on /project2/power, Job B weight is greater than Job A weight.

[82.]   In practice there can be thousands of jobs and thousands of nodes in a farm, and the job sort routine is run for all jobs and nodes in the farm; optionally, before sorting, the jobs and nodes can be filtered to remove specific jobs **and/or workers**, e.g., jobs that have too high a priority to interrupt, or all engaged workers.

[89.]   **EDQS job type data architecture.** The EDQS job type data architecture keeps logically associated data and code items together in a job type directory (aka folder), which greatly simplifies maintenance issues and also provides operational advantages. The minimum set of job type data items in a given job type element are:

        name of job type element ("descriptor" datum),

        name of execution file for processing the job ("executor" datum),

        name of the submission dialogue file ("GUI name" datum),

        name of job iconic representation file ("icon" datum),

        submission command line, including command line arguments ("commander" datum),

        binding scripts,

        names of associated libraries, and

        name of index file (described below).

In the preferred embodiment of the EDQS invention, each job type element is a container, which

in turn is a subcontainer in each EDQS job container. The job type directory also contains code files identified in the job type element, e.g., execution file, submission dialog file, job iconic representation file, binding scripts, index files, some or all of the associated libraries; the job type directory is typically stored on a supervisor. **In general, the EDQS invention can be implemented using OOP, procedural programming, and other programming techniques known in the art, but the preferred embodiments of the invention use OOP. In OOP embodiments of the invention, a "job type element" is replaced by a "job type object", and a "job type directory" is replaced by a "job container" (aka "EDQS job container"). In procedural language embodiments of the invention, a job type directory contains a job type element and code files identified in the job type element.**

[90.]   Consolidating the job type data and code items in a single directory simplifies maintenance, since a list of data and code items in **a given** directory serves as a checklist for maintenance. For example, if the execution file is modified, both the GUI name and the commander often need to be updated, which is easier if the GUI name and commander are located in the same directory as the execution file. In the preferred embodiment, a job type element, populated with data, is an OOP job type object.

[91.]   Use of EDQS job type elements and job type directories **(or in the OOP paradigm, job type objects within job containers)** not only facilitates maintenance (e.g., distribution of maintenance releases and bug fixes) and interaction between end-users and system administrators with software designers and developers, but enables an improved, uniform data architecture, especially when using distributed production facilities. A job type directory can be easily compressed, moved among production facilities, and installed in identical logical locations in each production facility to enable uniform network addressing within each production facility and among production facilities.

[95.]   "Location abstraction" means that a job type directory (a job type directory contains only one job type element, but, as explained above, the job type element can contain platform differentiation) can be hosted and used in several different ways. A job type directory can either

be network mounted (hosted on a server in the network used by a farm) and accessed by a worker daemon, or a job type directory can be transmitted to the worker at run time. Because all information required to process a given job is provided by a ~~job object and~~ job type directory **(or in OOP embodiments, a job container)**, a complete instruction to the worker daemon can be just a job ~~object~~ **type directory**, which includes a job type element~~, object and job type directory~~**(or in OOP embodiments, a complete instruction to a worker can be just a job container, which includes a job type object)**. Where job type directories are network mounted, a complete instruction to a worker daemon can be just a ~~job object, which~~ **message that** includes a job type element, and one or more job type directory paths; the job type directory path enables the worker daemon to look in several pre-established places for the job type directory. Providing alternative job type directory paths helps to minimize delays arising from network congestion, a server being off-line, or other causes.

[103.] EDQS DSL is also used for inter-process messaging within the network used in a distributed farm ("farm network"). Input from users who submit jobs for processing, and input from system administrators, is collected in graphical user interface ("GUI") and translated into EDQS DSL. The user GUI typically collects information related to a job ("job attributes") from a user or system administrator, translates the job attributes into a job request (a job request comprises one **or** more EDQS DSL statements), and sends the statements to the supervisory daemon. As explained below, the supervisory daemon builds a series of EDQS DSL statements that the supervisory daemon executes sequentially, subject to interrupts. In a preferred embodiment of the invention, each EDQS DSL statement to be executed is a record in a database.
[104.] EDQS takes advantage of the power and flexibility of object oriented language to "glue" code and data together, as well as to glue applications together by passing commands and data between applications. Third generation languages, such as C++, or scripting languages, such as Perl, include the ability to use complex data structures, such as scalars, associative arrays (aka hashes), and ~~arrays .~~**arrays.** These data structures, when bound to queuing system supplied libraries, allows the queuing system to convey abstract data rather than command lines and command line parameters. The preferred language used to build the EDQS DSL is C++, together with a simple "C" library for script language binding. XML can also be used to build

EDQS DSL. Different EDQS DSL statements contain data structures to submit a job, dispatch at job to a worker, to manage workers, **and** generally to serve all messaging requirements in the EDQS queuing system. EDQS DSL statements are generated by a process, the EDQS DSL generator, and EDQS DSL statements are interpreted by a second process, the EDQS DSL interpreter. There is ~~a~~**an** EDQS DSL generator and interpreter ~~for~~**hosted on** each platform used in a farm, ~~which~~**EDQS DSL generator and interpreter** are also built using an object oriented language, preferably C++. The appropriate EDQS DSL message generator and message interpreter are installed, at a minimum, on each supervisor, worker, and GUI-equipped computer. The generator converts platform-specific commands into a data structure that can be interpreted by other types of platforms in the farm. The C or XML library contains the code used by the EDQS DSL generator and interpreter to generate and to interpret, respectively, EDQS DSL statements.

[106.] The EDQS DSL interpreter that reads the EDQS DSL statement is compatible with the platform on which it runs (Unix, Windows, etc.), and sends to its local operating system the correct command line statement. In the illustration, the interpreter on a Unix machine ~~interpret~~**interprets** the EDQS DSL statement and send the command "ls –la/tmp" to the operating system. The sender of the illustrated message does not need to know the type of platform on the receiver. The command, ls –la /tmp, seems to be a valid statement to send to another resource. However, because 'ls' doesn't exist on a Windows machine, the command will fail.   With the introduction of the data structure used in the EDQS DSL statement, EDQS DSL interpreter on a Windows machine sends the proper command to "list all files, long format, in the current directory" to the Windows operating system. The EDQS DSL statement on the right is already parsed and ready for use by all resources in a farm, e.g., by all users for job submission, by the supervisor for interaction with users and workers, and by all workers. Since the contents of a EDQS DSL statement are plain text, and in a defined data structure, it is possible to recover the data very simply and quickly.

[108.] There are many advantages to using EDQS DSL. First, EDQS DSL eliminates quirky limits on command line parameters. Because the data structure in an EDQS DSL statement is

handled outside of the command line infrastructure, it is possible to transmit anything, including entire files, as part of a-**an** EDQS DSL statement. EDQS DSL statements can include information about the job itself, such as any processing requirements and restrictions, and other important information that normally is the subject and several queries and replies in existing art queuing systems.

[109.] Another advantage of EDQS DSL is the ability to include additional code and/or data in a statement. For example, an executable may not be a command line statement, but a dynamic link library or a script that requires other executables or data. These other executables and data can be included to populate the data structure of a-**an** EDQS DSL statement.

[110.] **EDQS event/callback architecture.** The primary tasks of a management daemon in existing art queuing systems is to match jobs and hosts, dispatch jobs to suitable hosts, respond to requests from users for job status reports, remedy failures of hosts to process jobs, and route submitted and completed jobs. As described above, existing art queuing systems typically use a "collect and analyze" (aka "periodic dispatch") sort and dispatch method that creates, and periodically recreates, a master job **dispatch** queue. The management daemons in existing art queuing systems also use polling extensively to collect information upon which conditional actions are based. For instance, each host is polled to learn whether processing of a job is ongoing or completed. Rather than use a polling and periodic process model of controlling execution of a management daemon (usually with a single-threaded sort and dispatch routine), the EDQS invention uses "events", "triggers," "callbacks", and a multi-threaded supervisory daemon, each of which threads executes "callbacks". A "callback" is a combination of an enabled "trigger" and executable code. A "trigger" is enabled when one or more "events" satisfy conditions defined for enablement of that trigger. An "event" is an expression that contains one or more Boolean parameters linked to states of specified variables e.g., the receipt by the supervisory daemon of certain messages from users or workers. Some of the important events defined in a preferred embodiment of the EDQS are described below. The receipt of a New Job message, for instance, is an event that satisfies the condition required for the Start Job trigger, which in turn causes the Start Job callback to be executed. The Start Job callback involves, among other things, an exchange of (in the preferred embodiment, EDQS DSL) messages with a worker that will process the job associated with the New Job message. Thus, the supervisory

daemon is event-driven, and uses an "event/callback" method of job allocation and management. The supervisory daemon uses certain tables contained in each job container: each job container always contains a callback table and a callback helper table, and may contain an agenda table and a subjob table. The subjob table is used to manage the individual instances of a job in each individual worker to which a part of the job is assigned, for instance, when a job to render a series of animation frames is processed by more than one worker. The agenda table enables the system to match the granularity of the subjob to a given application on a given worker. As an example, the agenda table facilitates executing a single frame of a computer graphics animation on a rendering application's most efficient computational element, which may be several processors in a single computer chassis. Using an agenda table shortens runtime by feeding more elements to faster workers. The executor assigns each component of a subjob that is assigned to a computational element an "agenda ID" or "agenda id" to enable tracking of the subjob component.

[112.] The preferred embodiment includes a callback table and a callback helper table in the job container itself to minimize lookup time and to maximize flexibility compared with using a table external to the job container; using tables external to the job container is an ~~alterative~~ **alternative** embodiment. "Callback table" and "callback helper table" are defined below, after a description of events and "triggers". The EDQS invention also contains a global callback table and a global callback helper table to handle callbacks that are not associated with a specific job. [113.] Generally speaking, an event is a change in the state of a hardware or software variable that can be reduced to a binary value, as such states are defined in "evtags". Evtags are easily tailored **to** meet user needs. Each evtag comprises three or more alphanumeric fields, each called an "evfield", and uses a defined syntax. In the preferred embodiment of the EDQS invention, three minimum evfields are used: <evname>-, <evtype>-, and <evcontext>; optional evfields are denoted <evextra1>, <evextra2>, etc. A given <evname> value can be coupled with various <evtype> values, and each <evtype> can be coupled with various <evcontext> values, to create the evtags used by the EDQS invention. Equivalent methods known in the art to create a tiered taxonomy of variables and states can be used. Each evfield in an evtag has a specific meaning and purpose. The term "event" technically means the event defined in the relevant

evtag. **As explained below, an evtag is a mandatory part of a "trigger definition".** The format of each evtag is as follows:

<evname>-<evtype>-<evcontext>[-<evextra1>][-<evextra2>]...[-<evextraN>]

[115.] In the lists of the evfields below, "the job" means the job identified in the <evcontext> field of the evtag. The following <evname> evfields, and associated definitions, are used in the preferred embodiment.

memorylimit – the job has attempted to use more memory than the worker processing the job has allocated to the job

timelimit – the job has been running for longer than the processing time assigned to the job

~~submit – the job is submitted~~

submit – the job is submitted

failed – the job failed to complete its processing

complete – the job completed processing successfully

killed – a user (usually the submitting user) has killed the job

blocked – the job is blocked, i.e., in a state where it can not be set to "running" until it is unblocked.

running - processing of the job has started

removed – the job is removed from the job list.

suspended – the job is suspended, i.e., the job has signaled its active subjob processes to temporarily stop processing

waiting – the job is waiting for an available worker to execute a subjob ~~on.~~ **on**

assigned – the job is assigned to a worker

done – the job is in one of three completion states: complete, failed, killed

modified – the job is modified

unknown – the job is unknown

timeout – the job has timed out

[119.] FIG. 13 illustrates job status variables and values. **As explained in more detail below in connection with Figure 20, each worker daemon exchanges messages with its executor(s) and with the worker daemon's supervisory daemon; messages from the executors relayed by the worker daemon to the supervisory daemon contain, inter alia, the Boolean state of evfield(s) sent by an executor upon a change of state of evfield(s). The supervisory daemon uses these messages to track workers and job status, and to determine if evtags and other statements in trigger definitions evaluate as Boolean "true" or "false".**

FIG. 14 illustrates evtype variables and values.

[121.] In the preferred embodiment, a trigger occurs when the Boolean expression defined for that trigger is true. For example, a trigger definition to start job D upon the completion of jobs A, B, and C would be:

evtag**1** := complete-job-jobA and complete-job-jobB and complete-job-jobC

Start JobD = evtag1

<trigger> := Start JobD

[123.] A callback is a trigger associated with code relevant to the trigger. In a preferred embodiment, a callback is an OOP object comprising a trigger and executable code specific to that trigger. The "trigger" is the Boolean expression **(i.e., the trigger definition)** that, when it evaluates to true, causes the code associated with the trigger to be executed. The callback table in a given job container contains (i) all triggers to be used for that job and (ii) the executable code associated with each trigger. The "callback helper table" facilitates callback referencing and contains, among other things, the Boolean state of all events used in all the "trigger definitions" contained in the callback table. This enables the supervisory daemon to quickly identify the events that have a Boolean state of true and to find the callbacks associated with those events through trigger definitions. The callback helper table contains pointers to such callbacks. In the preferred embodiment, each record in a job's callback table is a single callback.

[125.] The data structure of a callback requires the **relevant** trigger and code components. Multiple variants of the code component, differing by platform, can be provided. Additional components of a callback are optional, as follows:

Language – Description of the type of machine code or bytecode, or of the source code scripting language, in which the one or more code variants are written

Count – Number of times the code has been executed

Max – Maximum number of times the code should be executed

Job – the job a callback is part of

[126.] FIG. 20 is a flowchart of the steps in used by a process, the "event/callback layer", in the supervisory daemon to detect an enabled trigger and to execute the callback code contained in the callback containing that trigger. A network/socket/handler, a process on the supervisor (which uses message exchange methods known in the art) receives a network message, identifies the correct message handler, **and** notifies the event/callback layer. When a job is submitted, the user interface places in that job container's callback table the callbacks (i.e., the pairings of trigger definition and executable code associated with a given trigger) that the user intends to execute with the job. When the supervisory daemon (more accurately, the thread of the supervisory daemon that is active on the supervisor when a New Job message arrives at the supervisor) receives a given New Job message, the supervisory daemon analyzes (using the steps shown in FIG. 20) the trigger definitions of each callback in the job's callback table as part of the supervisory daemon's validation of a newly submitted job ("new job validation"). The trigger analysis will identify all the evtags that the supervisory daemon must track between submission and completion of the job. The supervisory daemon maintains a database that contains fields for all events the supervisory daemon must track (including fields for state values for each such event) and for the jobs that use a given event in their trigger definitions ("event database"). Entries in the database record of a given database include the memory location of the callback itself for interpretation by the appropriate script interpreter, execution by a dynamic library loader (or equivalent), or compiling by a virtual machine, as appropriate for a given platform.

[127.] After analysis of each trigger definition, the supervisory daemon creates a new job record in the event database and populates it with values for the events used in that job's callback table. The supervisory daemon also sets up in the job container's callback helper table a checklist for the relevant callback.   Upon detection of an event, the supervisory daemon scans the event

database for jobs that use that event (i.e., have the event in a job's callback table), and evaluates the trigger definitions of such jobs that contain that event (technically, **such jobs that contain** that evtag) to determine if the just detected event enabled a given trigger. In cases in which one or more earlier events failed to enable a trigger (i.e., the trigger definition evaluated as Boolean false), a later event can supply the state change needed to enable the trigger; the earlier events are tracked as state values in the event database. In another preferred embodiment, state values of events are not tracked in the event database but in the callback helper table in the relevant job container. The trigger definitions in all callback tables of jobs that have been submitted but are not yet completed are searched for occurrences of an event that has been detected and then such trigger definitions are evaluated with reference to the new event and to the event state values in the callback helper table (or event database, depending upon the embodiment). Some events, such as a worker with attributes required for a given job becoming unavailable after previously being available, can cause a previously true state value in the event database or the callback helper table, depending upon the embodiment, to be reset to a Boolean false upon the supervisory daemon's receipt of a message that the worker is unavailable. Associating callbacks with jobs also simplifies the supervisory daemon's task of removing entries in the events database of jobs that are "done" (completed, failed, or killed).

[131.] After analysis of each trigger definition, the supervisory daemon creates a new job record in the event database and populates it with values for the events used in that job's callback table. The supervisory daemon also sets up in the job container's callback helper table a checklist for the relevant callback.   Upon detection of an event, the supervisory daemon scans the event database for jobs that use that event (i.e., have the event in a job's callback table), and evaluates the trigger definitions of such jobs that contain that event (technically, **such jobs that contain** that evtag) to determine if the just detected event enabled a given trigger. In cases in which one or more earlier events failed to enable a trigger (i.e., the trigger definition evaluated as Boolean false), a later event can supply the state change needed to enable the trigger; the earlier events are tracked as state values in the event database. In another preferred embodiment, state values of events are not tracked in the event database but in the callback helper table in the relevant job container. The trigger definitions in all callback tables of jobs that have been

submitted but are not yet completed are searched for occurrences of an event that has been detected and then such trigger definitions are evaluated with reference to the new event and to the event state values in the callback helper table (or event database, depending upon the embodiment). Some events, such as a worker with attributes required for a given job becoming unavailable after previously being available, can cause a previously true state value in the event database or the callback helper table, depending upon the embodiment, to be reset to a Boolean false upon the supervisory daemon's receipt of a message that the worker is unavailable. Associating callbacks with jobs also simplifies the supervisory daemon's task of removing entries in the events database of jobs that are "done" (completed, failed, or killed).

[144.] The EDQS messaging architecture can be used as a standalone component in a queuing system. The EDQS messaging architecture requires the messages described in Fig. 11, or substantial equivalents of them, but does not require a given type of searching, sorting, or dispatch routine, EDQS job routing, EDQS event/callback architecture, EDQS job type data architecture, ~~the EDQS messaging architecture,~~ or EDQS DSL.

[146.] EDQS job type data architecture can be used as a standalone component in a queuing system, for instance using job type directories and/or job type elements **(or their equivalents in an OOP paradigm)** containing a descriptor datum, an executor datum, a GUI name datum, and icon datum, a commander datum, binding scripts, names of associated libraries, and name of index file. Using such job type elements, especially together with job type directories, greatly reduces maintenance (e.g., distribution of maintenance releases and bug fixes) overhead in distributed production facilities.